NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

STATIC SCHEDULING OF CONDITIONAL BRANCHES IN PARALLEL PROGRAMS

by

Robert T. George

December 1996

Thesis Advisor: Theodore Lewis

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE December 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Static Scheduling of Conditional Branches in Parallel Programs(U)	5. FUNDING NUMBERS
6. AUTHOR(S) George, Robert Tyler	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES) Army Research Laboratory Information Sciences and Technology Directorate AMSRL-IS-PA 2800 Powder Mill Road Adelphi, MD 20783-1193	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The problem of scheduling parallel program tasks on multiprocessor systems is known to be NP-complete in its general form. When non-determinism is added to the scheduling problem through loops and conditional branching, an optimal solution is even harder to obtain. The intractability of obtaining an optimal solution for the general scheduling problem has led to the introduction of a large number of scheduling heuristics. These heuristics consider many real-world factors, such as communication overhead, target machine topology, and the trade-off between exploiting the parallelism in a parallel program and the resulting scheduling overhead.

We present the *probabilistic merge heuristic* -- in which a unified schedule of all possible execution instances is generated by successively scheduling tasks in order of their execution probabilities. When a conditional task is scheduled, we first attempt to merge the task with the time slot of a previously scheduled task which is a member of a different execution instance.

We have found that the merge scheduler produces schedules which are 10% faster than previous techniques. More importantly, however, we show that the probabilistic merge heuristic is significantly more scalable -- being able to schedule branch and precedence graphs which exceed 50 nodes.

14. SUBJECT TERMS Parallel Processing, Scheduling	15. NUMBER OF PAGES 64
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

STATIC SCHEDULING OF CONDITIONAL BRANCHES IN PARALLEL PROGRAMS

Robert Tyler George
Army Research Laboratory
B.S., Virginia Polytechnic and State University, 1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1996

Author: [illegible]
Robert Tyler George

Approved By: [illegible]
Theodore Lewis, Thesis Advisor

[illegible]
Dave Pratt, Second Reader

[illegible]
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The problem of scheduling parallel program tasks on multiprocessor systems is known to be NP-complete in its general form. When non-determinism is added to the scheduling problem through loops and conditional branching, an optimal solution is even harder to obtain. The intractability of obtaining an optimal solution for the general scheduling problem has led to the introduction of a large number of scheduling heuristics. These heuristics consider many real-world factors, such as communication overhead, target machine topology, and the trade-off between exploiting the parallelism in a parallel program and the resulting scheduling overhead.

We present the *probabilistic merge heuristic* -- in which a unified schedule of all possible execution instances is generated by successively scheduling tasks in order of their execution probabilities. When a conditional task is scheduled, we first attempt to merge the task with the time slot of a previously scheduled task which is a member of a different execution instance.

We have found that the merge scheduler produces schedules which are 10% faster than previous techniques. More importantly, however, we show that the probabilistic merge heuristic is significantly more scalable -- being able to schedule branch and precedence graphs which exceed 50 nodes.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem [1]. This definition is broad enough to include parallel supercomputers that contain hundreds or thousands of processors, networks of workstations, multiple processor workstations, and embedded systems. Parallel computers present a particularly interesting computer architecture because they offer the potential to concentrate enormous computational resources -- processors, memory, I/O bandwidth, etc. -- to solve computationaly expensive problems.

Parallelism has sometimes been viewed as a rare and exotic subarea of computing, interesting but of little relevance to the average programmer. Recent trends in applications, computer architecture, and networking shows that this view is no longer tenable. Parallelism is becoming ubiquitous, and parallel programming is becoming central to the programming enterprise.

A more important factor affecting the acceptance of parallel computing architecutures is the emergence of impending technical and economic obstacles which have begun to slow the pace of advances in semiconductor technology. In 1964, Gordon Moore observed that the number of transistors that semiconductor makers could put on a chip was doubling every year. By the late 1970's, however, the pace had slowed to a doubling of transistors every 18 months. Today, modern chips are being manufactured with 7 million transistors by aggressively exploiting the upper limits of optical lithography. If semiconductor are to continue to scale according to Moore's law, exotic technologies such as x-ray lithography will be required. More importantly, the costs of building semiconductor plants has also scaled linearly -- doubling every 3 years. A modern semiconductor facility currently costs $1 - $3 billion. Clearly, it may not be long before the semiconductor industry plateaus, the pace of transistor integration declines, and manufacturing costs begin to soar. Parallel processing provides an execellent alternative to the current reliance on the ever increasing integration of modern semiconductors.

B. SCHEDULING PARALLEL TASKS

Scheduling the tasks which comprise a parallel program is a classical field with several interesting problems and results. A scheduling problem emerges whenever there is a choice as to the order in which a number of tasks can be performed, and/or in the assignment of tasks to servers for processing. Such a scheduling problem may involve jobs that need to be processed in a manufacturing plant, bank customers waiting to be served by tellers, aircraft waiting for landing clearance, or program tasks that need to be run on a parallel or distributed computer. Clearly, there are fundamental similarities among scheduling problems regardless of the the nature of the tasks, and the environment.

In the era of parallel and distributed computing, the scheduling problem has begun to gain the attention of many researchers. A computer program can be viewed as a collection of tasks which may run serially or in parallel. The goal of scheduling tasks on a parallel computer is to determine an assignment of tasks to processing elements, and an order in which tasks are to be executed, in order to optimize some performance criteria. As a result, an optimal schedule will determine both the allocation of tasks to processors, and the execution order the tasks. If there are no precedence relations among the tasks forming a program, this problem is known as a *task allocation* problem. Task allocation has been studied extensively for the past two decades and is a somewhat different problem than task scheduling.

Task scheduling is one of the most challenging problems in parallel and distributed computing. It is known to be *NP-complete* in its general form as well as in several restricted cases [2]. Researchers have studied restricted forms of the scheduling problem by constraining either the task graph representing the parallel program, or the parallel computer model. When communication between tasks is not considered, a *polynomial time* algorithm can be found for scheduling tree-structured task graphs, where all tasks execute in unit time. Such special cases, although representing optimal solutions, do not accurately represent real-world systems. In an attempt to solve the scheduling problem in the general case, a number of heuristics have been introduced. A heuristic, by definition, does not guarantee an optimal solution to the problem, but attempts to find near-optimal solutions most of the time.

C. THE SCHEDULING PROBLEM

The *scheduling problem* has been described in a number of different ways in different fields. Job sequencing in production management, a classical problem from operations research, has influenced most of what has been written about this problem. Manufacturing processes often involve several operations to transform raw material into a finished product. The problem is to determine sequences of operations that are preferred according to certain (economic) criteria. The problem of generating these preferred sequences is referred to as the *sequencing problem.* Over the years, several methods have been used to solve the sequencing problem, including complete enumeration, heuristic rules, integer programming, and sampling methods. It is clear that complete enumeration is impractical, and optimal solutions cannot be obtained in real time. As a result, heuristic methods have been used to provide solutions to the most general case of the problem.

In general, the scheduling problem assumes a set of resources, and a set of consumers serviced by these resources according to a certain policy. Based on the nature of and the constraints on the consumers and the resources, the problem is to find an efficient policy for managing the access to and the use of the resources by various consumers to optimize some desired performance measure such as *schedule length.* Accordingly, a scheduling system can be considered as consisting of a set of consumers, a set of resources, and a scheduling policy. A task in a computer program, a job in a factory, or a customer in a bank are examples of consumers. A processing element in a computer system, a machine in a factory, or a teller in a bank are examples of resources. *First-come-first-served* is one example of a heuristic scheduling policy. Natually, scheduling policy performance varies with different circumstances. While first-come-first-served may be appropriate in a bank environment, it may not necessarily be the best policy to schedule jobs on a factory floor.

Performance and efficiency are two characteristics used to evaluate a scheduling algorithm. We should evaluate a scheduling system based on the goodness of the schedule produced, and the efficiency of the policy. In other words, we are concerned with both the quality of the generated schedule and the efficiency of the scheduler itself. The resulting schedule is judged by the performance criteria we are trying to optimize. For example, if we are optimizing the completion time of a program, the less time the schedule takes, the better the schedule. Both the scheduler and the scheduling policy can be evaluated based on their respective time complexities. If two policies produce schedules of equal quality, then the simpler one is clearly better.

In this thesis, we are concerned with scheduling program tasks on parallel and distributed computers. The tasks are the consumers and will be represented using directed acyclic graphs called *task graphs*, while the processing elements are the resources and their interconnection networks will be represented using undirected graphs. The scheduler generates a schedule using a timing diagram call the *Gantt chart* to illustrate the allocation of the parallel program tasks onto the target machine processors. The Gantt chart consists of a list of the processors in the target machine and, for each processor, a list of tasks allocated to that processor, ordered by their execution time, including task start and finish times.

II. THE TASK SCHEDULING MODEL

A. BACKGROUND

In this section we describe a general model to formulate the scheduling problem. The models here are deterministic, in the sense that all information governing the scheduling decisions is assumed to be known in advance. In particular, the task graph representing the parallel program and the target machine is assumed to be available before the program starts execution. The non-deterministic scheduling problems analyzed in this thesis can be represented as special cases of this model. Loops, however, cannot be represented in parallel programming models using this system.

There are four components in any scheduling system:

1. the target machine
2. the parallel program tasks
3. the generated schedule
4. the performance criteria

We will review each of these components and show how the program and target machine parameters can be used to estimate execution times and communication delays.

B. TARGET MACHINE

The target machine is assumed to be made up of m heterogeneous processing elements connected using an arbitrary interconnection network. Each processing element can run one task at a time, and all tasks can be computed by any processing element. Formally, the target machine characteristics can be described as a system $(P, [P_{ij}], [S_i], [I_i], [B_i], [R_{ij}])$ as follows:

- $P = \{P_i, \ldots, P_m\}$ is a set of processors forming the parallel architecture.
- $[P_{ij}]$ is an $m \times m$ interconnection topology matrix.
- S_i, $1 \le i \le m$, specifies the speed of processor P_i.

- I_i, $1 \le i \le m$, specifies the start-up cost of initiating a message on processor P_i.
- B_i, $1 \le i \le m$, specifies the start-up cost of initiating a process on processor P_i.
- R_{ij} is the transmission rate over the link connecting two adjacent processors P_i and P_j

The connectivity of the processing elements can be represented using an undirected graph called the *target machine graph.* Processors are occasionally referred to simply by their indices (e.g., 1 may be used rather than P_1), especially when target machine nodes are more conveniently labeled with integers.

C. PARALLEL PROGRAM TASKS

A parallel program is modeled as a partially ordered set (poset) $(T, <)$, where T is a set of tasks. The relation $u < v$ implies a data dependency between tasks u and v. The computation of task v depends on the results of the computation of task u, so task u must be computed before task v, and the result of computation of task u must be known by the processor computing task v. The characteristics of a parallel program can be defined as the system $(T, <, [D_{ij}], [A_i])$ as follows:

- $T = \{t_1, \ldots, t_n\}$ is a set of tasks to be executed.
- $<$ is a partial order defined on T which specifies operation precedence constraints. That is, $t_i < t_j$ signifies that t_i must be completed before t_j can begin.
- $[D_{ij}]$ is an $n \times n$ matrix of communication data where $D_{ij} \ge 0$ is the amount of data required to be transmitted from task t_i to task t_j, $1 \le i \le n$, $1 \le j \le n$.
- $[A_i]$ is an n vector of the amount of computations, i.e., $A_i > 0$ is the number of instructions required to execute t_i, $1 \le i \le n$.

The partial order $<$ is conveniently represented as a directed acyclic graph called a *task graph.* A directed edge (i,j) between two tasks t_i and t_j specifies that t_i must be completed before t_j can begin. Figure 1a shows an example of a task graph consisting of eight nodes (n=8), where each node represents a task. The number shown in the upper portion of each nodes is the node number, the number in the lower portion of a node i represents the parameter A_i (the amount of computation needed by task t_i), and the number next to an edge (i,j) represents the parameter D_{ij}. (the amount of communication between node i and node j). For example, $A_1 = 5$, $D_{15} = 10$. Tasks are often referred to simply by their indices (e.g., *1* may be used rather than t_1), especially when graph nodes are more conveniently labeled with integers.

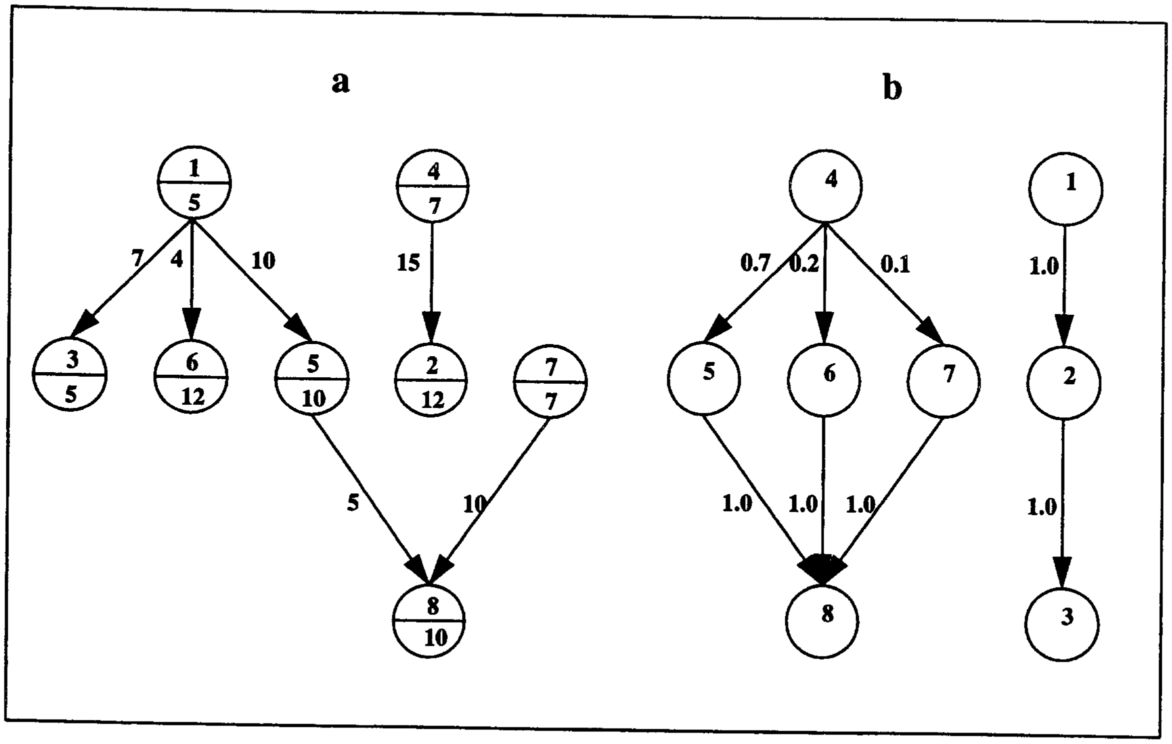

Figure 1: (a) Precedence graph (b) Branch graph

1. Modeling Conditional Branching

A parallel program can be viewed as a set of tasks and a flow of control and data through these tasks. A parallel program can be represented using two directed acyclic graphs -- the *branch graph*, $G = (T, E_b)$, and the *precedence graph*, $H = (T, E_p)$, where T is the set of n verticies representing the program tasks, E_b is the set of branch edges, and E_p is the set of precedence edges. An *execution instance* of a parallel program is defined as a possible set of tasks that are selected for execution at one time for some input. Associated with each branch edge *(u,v)* is *P(u,v)*, the probability of having v in the same execution instance with u. The summation of the probabilities associated with the edges, leaving a node in the branch graph, is always one.

A branch graph consists of a collection of connected components. Each component of the branch graph is, in turn, a *fan graph*. This implies that each node has n independent nodes with one common parent and one common child. Fan graphs are typically used to express conditional branching in structured programming, where the independent nodes represent different alternatives in a branching statement. Figure 1b shows an example of of a branch graph consisting of 8 nodes which represent a parallel program. The number inside each node is the node title and the number next to an edge *(i,j)* represents the probability *P(i,j)*.

A precedence edge *(u,v)* implies that task v cannot begin execution until after task u has completed execution. This edge might also represent data flow between tasks u and v. Associated with the edge *(u,v)* is the data size *D(u,v)*. Associated with each task u is the number of instructions to be executed, *INS(u)*. Thus, task graph in Figure 1a can also be referred to as a precedence graph. For example, t_5 cannot begin execution until after t_1 has completed execution.

2. Execution and Communication Cost

Given a parallel program model and the description of the target machine that will execute the program, task execution time and the communication delay can be obtained as follows:

T_{ij}: the execution time of task i when executed on processor j. It can be computed as follows: $T_{ij} = \frac{A_i}{S_J} + B_j$

$C(i_1,i_2,j_1,j_2)$: the communication delay between task i_1 and i_2 when they are executed on processing elements j_1 and j_2, respectively. This reflects the target machine performance parameters as well as the size of the data to be transmitted and can be computed as follows:

Suppose that j_1 and j_2 are two adjacent processing elements, then the communication delay of a message sent from task i_1 running on j_1 to task i_2 running on j_2 over a free link is:

$$C(i_1, i_2, j_1, j_2) = \frac{D_{i_1 i_2}}{R_{j_1 j_2}} + I_{j_1}$$

Typically, more than one message can be sent from one processor to another using the same link. This implies that contention delay must be considered. If we assume that we can estimate the contention delay on the link connecting processors j_1 and j_2 as CD_{j1j2}, the above formula becomes:

$$C(i_1, i_2, j_1, j_2) = \frac{D_{i_1 i_2}}{R_{j_1 j_2}} + I_{j_1} + CD_{j_1 j_2}$$

Finally, we must consider the case where the source and destination processors are not adjacent. Suppose that a message from j_1 to j_2 is sent through the path j_1, k_1, k_2,..., k_z, j_2. It can be noted that the number of hops is $z + 1$. The communications delay now becomes:

Although this involves a formidable calculation, consider the case where all processing nodes have the same I/O co-processors and therefore have identical message

$$C(i_1, i_2, j_1, j_2) = \frac{D_{i_1 i_2}}{R_{j_1 k_1}} + I_{j_1} + CD_{j_1 k_1} + \frac{D_{i_1 i_2}}{R_{k_z j_2}} + I_{k_z} + CD_{k_z j_2}$$
$$\sum_{i=1}^{z-1} \left(\frac{D_{i_1 i_2}}{R_{k_i k_{i+1}}} + I_{k_i} + CD_{k_i k_{i+1}} \right)$$

initiation times I, and the transmission rate R is uniform over the interconnection network. The communications delay then reduces to:

$$C(i_1, i_2, j_1, j_2) = \left(\frac{D_{i_1 i_2}}{R} + I \right) \cdot (z+1) + CD_{j_1 j_2}$$

D. THE SCHEDULE

A schedule of the task graph $G = (T,A)$ on a target machine made up of m processors is a function f that maps each task to a processor with an assigned start time. Formally, $f | T \to \{1, 2, \ldots, m\} \times [0, \infty)$. If $f(v) = (i,t)$ for some $v \in T$ we say that task v is scheduled to be processed by processor p_i starting at time t. Note that there exists no $u, v \in T$ such that $f(v) = f(u)$ -- two tasks cannot run on the same processor at the same time. If $v < u$ and $f(v) = (i,t_1)$, $f(u) = (j,t_2)$, then $t_1 < t_2$. Clearly, a schedule f is feasible if it preserves all precedence relations and communications restrictions. The Gantt chart provides an informal notion of the schedule, where the start and finish times for all tasks can be easily shown.

E. PERFORMANCE CRITERIA

In light of the description of the scheduling problem, we would like to find efficient algorithms for scheduling the tasks on the available processors to optimize some desired performance measure. Several such performance criteria include balancing the computational load across the target processors and minimizing the completion time of the schedule. In this thesis, our goal has been chosen to minimize the total completion time of a parallel program. This performance measure is known as the *schedule length*, or *maximum finishing time*. Schedule length can be described as follows. Given a task graph $G = (T,A)$ and a schedule f on m processors, the length of schedule f of G is the maximum finishing time of any task in G. Formally, $length(f) = t_{max}$ where $t_{max} = maximum\{t + T_{ij}\}$ where $f(i) = (j,t)$ $\forall i \in T, 1 \leq j \leq m$. Recall that T_{ij} is the execution time of task i on processor j.

III. LIST SCHEDULING

A. BACKGROUND

As shown in the previous chapters, optimal schedules can be obtained in very restricted special cases. These special cases are far different from real world situations. One may question how we can restrict a parallel computer to have only two processors in the era of massively parallel architectures with hundreds and thousands of processors, or how can we neglect the effect of communication delay in distributed-memory systems. To provide useful solutions to the scheduling problem, restrictions on the parallel program and target machine representations must be relaxed. Recent research in this area has emphasized heuristic approaches. A heuristic algorithm produces an answer in less than exponential time, but does not guarantee an optimal solution [3]. Therefore the *near optimal* solutions obtained by a heuristic approach the optimal solution most of the time. Intuition is most often used to derive heuristics that make use of scheduling parameters that affect the system in an indirect way. One heuristic is said to be better than another if solutions approach optimality more often, or if less time is spent obtaining a near-optimal solution.

B. LIST SCHEDULING

The most common class of scheduling heuristics is *list scheduling*. In list scheduling each task in a parallel program is assigned a priority, and a list of tasks is constructed in decreasing priority order. When a processor becomes available, the *ready task* with the highest priority is selected from the list and assigned to that processor. If more than one task has the same priority, a task is selected arbitrarily. Algorithm 1 presents a generic procedure of list scheduling.

Algorithm 1

1. Each node in the task graph is assigned a priority. A priority queue is initialized for ready tasks by inserting every task that has no immediate predecessors. Tasks are sorted in decreasing order of task priorities.

2. As long as the priority queue is not empty do the following:

- Obtain a task from the front of the queue.
- Select an idle processor to run the task.
- When all the immediate predecessors of a particular task are executed, that successor is ready to be inserted into the priority queue.

Thus, nodes are selected in order according to their priority assignment. The *level* and *co-level* of a task are examples of task priority. We define the following:

- **Path Length.** The length of a path in a task graph is the summation of the weights of all nodes along the path including the initial and final node.
- **Level.** The level of a node in a task graph is defined as the length of the longest path from the node to an exit node (an exit node is the one with no successors).
- **Co-Level.** The co-level of a node in a task graph is defined in the same way as a level except that lengths are measured from the starting points of the task graph rather than from the exit node.

In their paper in 1974, Adam, Chandy, and Dickson conducted an extensive empirical performance study of five list-scheduling heuristics as follows [12]:

- **HLFET** (Highest Levels First with Estimated Times). A list schedule in which the priority of a task is its level.
- **HLFNET** (Highest Levels First with No Estimated Times). In this list schedule, all tasks are assumed to have the same execution time. The priority assigned to a task is its level computed under this assumption.
- **RANDOM**. A list schedule in which tasks were assigned priorities randomly.
- **SCFET** (Smallest Co-levels First with Estimated Times). In this list schedule, the priority of a task is the negative of its co-level.
- **SCFNET** (Smallest Co-levels First with No Estimated Times). This list schedule is the same as SCFET except all tasks are assumed to have the same execution time.

The results of the study showed that among all priority schedulers, level priority assignment generates the closest to optimal schedules. Highest level first (HLF), which is also known as critical path (CP), was shown to be superior to others since it provided schedules that are within 5 percent of the optimum in 90 percent of random cases. It also has been shown that using level priorities will produce an optimal schedule for a task graph represented by a tree [3]. In tree structured task graphs of equal task execution times, Hu introduced an optimal algorithm using level (or critical path) assignment [8].

It is import to note that these heuristics do not consider the cost of inter-task communications. Thus, they are most appropriate for shared-memory parallel processors where memory access costs are uniformly distributed among the computational elements. Distributed-memory parallel computers, in contrast, have communication costs which are determined by the nature of the inter-processor communications network. The question then arises as to how communication delays may affect list-scheduling heuristics. In the following section we discuss two problems that are introduced once inter-task communication is considered.

C. COMMUNICATION ISSUES

In this section we study some of the difficulties encountered when designing heuristic schedulers for a more general case, where communication delays are considered. The first problem is due to the trade-off between exploiting maximum parallelism and minimizing communication delay, while the second problem is due to the alteration of critical paths of the tasks in a task graph.

1. Parallelism Versus Communication Delay

Considering communication delays in making scheduling decisions introduces a key difficulty in scheduling parallel programs, the so called *max-min* problem. This problem is associated with the trade-off between taking advantage of maximal parallelism in the task graph and minimizing communication delay. If tasks are allocated to processors in such a manner as to maximize the amount of simultaneous execution of tasks without regard to the cost of message transmission, the result may be a program that runs slower on several processors than it does on a single processor. This case arises when communication costs are high compared to execution costs. Alternatively, when available parallelism is not completely exploited, available processors may be underutilized.

When there is no communication delay between tasks, all ready tasks can be allocated to all available processors. The will result in a reduction of the overall execution time of the task graph. This reflects the assumptions made by earlier schedulers which did not consider communication delays. If communications delays are to be included in the list scheduling heuristic, then scheduling must be based on both the inter-task communication delay and then time when each processor is ready for execution. It is possible for ready tasks with long communication delays be assigned to the same processor as their immediate predecessors. Hence, adding communication delay constraints increases the difficulty of arriving at an optimal schedule, because the scheduler must examine the start time of each node on each available processor to select the best one. It would be a mistake to increase the amount of parallelism available by simply starting each task as soon as possible. Distributing parallel tasks to as many processors as possible inevitably increases the communication delay which, in turn, increases the overall execution time. In short, there is a trade-off between taking advantage of maximal parallelism versus minimizing communication delay. This problem is called the max-min problem for parallel processing. It is the goal of current communication delay scheduling heuristics attempt to take advantage of parallelism, while reducing communication delays.

2. Level Alteration

Another important scheduling problem caused by the introduction of non-zero communication delays is due to the alteration of level priorities, and their impact on critical path calculation. Any heuristic that uses level priorities or critical path length faces this problem. The level of a node is defined as the length of the longest path from the node to the exit node. This length includes all node execution times and communication delays along the path. Unfortunately, the level priority does not remain constant when communication delays are considered, since the level of each node changes as the length of the path leading to the exit node changes. The path length varies depending on communication delay and the communication delay changes depending on task allocation. Communication delay is zero if tasks are allocated to the same processor and non-zero if tasks are allocated to different processors. The number of network hops between processors will also make a difference in computing the communication delay portion of the level. We call this the *level number problem* for parallel processor scheduling. As a further complication, when the target machine processors are not identical the execution time used in the computation of the level priority again becomes difficult to obtain because the execution time of a node depends on the speed of the processor that executes that node.

Some heuristics assume identical processors and compute a node's level as the summation of the node's computations along the path to the exit node, excluding the communication delay. A better approximation of level number may be obtained by iteration: schedule, then calculate node level, schedule, etc. The time complexity would be tremendously increased and the resulting level number would still be only an approximation. Hence, the use of level as priority for scheduling with communication delay is less accurate than that without communication delay.

El-Rewini and Lewis conducted several experiments to show the effect of using communication delays in calculating the level of a node in a task graph [13]. This scheduling heuristic involves adding the communication delay to the execution time when computing the level of a node. The results of the experiment suggest that for communication intensive applications the scheduler should consider communication delay in the scheduling algorithm's priority; however, for computation-intensive applications, priority scheduling is insensitive to communication delays of the application.

3. List Scheduling with Communication

In the previous sections, we examined list-scheduling heuristics when communication is ignored and showed that using the level of a task as its priority is near optimal most of the time. We also showed two problems that were introduced by considering communication delay in making scheduling decisions. We now consider heuristics where communication delay is considered. In this section, we give the program and machine assumptions and terminology that will be used in the rest of this chapter. We will also show how the level heuristic, originally introduced by Hu, is modified to handle communication. In this heuristic, which we will refer to as the general list heuristic, task selection criteria remain the same as described in Algorithm 1. The communication delay is included in computing the start time of a selected task when selecting a processor, but is ignored when the level of each task is computed. The processor containing the assigned task's immediate predecessors are considered first in an attempt to reduce communication delay by placing message source and destination tasks on the same processor.

4. Program and Machine Models

In this section, we define our program and target machine models in terms of the general model introduced in Chapter II. Recall that the target machine model was represented as the 6-tuple $(P, [P_{ij}], [S_i], [I_i], [B_i], [R_{ij}])$, while the parallel program tasks were represented as the 4-tuple $(T, <, [D_{ij}], [A_i])$. The assumptions are as follows:

Target Machine

- $P_{ij} = 1,\ 1 \le i, j \le m$, (a fully connected system)
- $S_i = constant,\ 1 \le i \le m$, (identical processing elements)
- $I_i = 0,\ 1 \le i \le m$, (no start-up cost for initiating a message)
- $B_i = 0,\ 1 \le i \le m$, (no start-up cost for initiating a task)
- $R_{ij} = constant,\ 1 \le i, j \le m$, (Same transmission rate)

Program Tasks

- There are no restrictions on the task graph representing the program.

Having identical processing elements, it follows that the execution time, T_{ij} of task i on processor j will be the same for all processing elements, which is equal to $\frac{A_i}{S_i}$

Assuming that $S_i = 1$, we can denote A_i as the execution time of task i, which will also be referred to as the *task size*. Having the same transmission rate for all links, it follows that the communication time for a message sent from task i_1 on processor j_i to task i_2 on processor j_2, $C(i_1, i_2, j_1, j_2)$, is equal to $\frac{D_{i_1 i_2}}{R_{j_1 j_2}}$. Assuming that $R_{j_1 j_2} = 1$ we can use $D_{i_1 i_2}$ as the communication delay.

5. Terminology

In this section, we define some of the terms that will be used to describe the heuristics in the following sections of the chapter.

- The *ready time* of a processor is the time when the processor has finished its assigned task and is ready for another task.
- The *message ready time* of a task is the time when all the messages to the task have been received by its processor. This time represents the largest communication delay of all the messages sent from the task's immediate predecessors. The immediate predecessor that causes the longest communication delay is called the *latest immediate predecessor* (**LIP**).
- The *ready queue* is a queue of ordered ready tasks. Tasks are ordered according to their levels; the task with the highest level is scheduled first. Tasks at the same level are ordered according to the number of immediate successors; the task with the greatest number of immediate successors is scheduled first
- The *assigned task* (**AN**) is the highest-priority task selected from the ready queue.
- The *idle time slot* is the time interval between the ready time of a processor and the assigned task's starting time.
- The *assigned processor* is the one chosen to execute the assigned task.
- The *first ready processor* (**PRF**) is the first processor in the set of all processors to become ready after the assigned task is scheduled on the

assigned processor.

6. General List-Scheduling Heuristic

In this section we give the details of the *general list heuristic* with communication. This heuristic tries to minimize the total elapsed time to execute all tasks by minimizing the finishing time of each assigned task. First the level of each task in the task graph is calculated and used as each task's priority. The ready queue is then initialized by inserting all tasks with no immediate predecessors into the ready queue. Then, the task at the top of the queue (the task with the highest priority, i.e., highest level) is assigned to a processor. The processor is selected by the selection routine `locate_p`. For the first task, any processor can be selected. The heuristic continues assigning tasks and updating the ready queue until all the tasks in the task graph are assigned. Each time a task is assigned, the assigned processor is marked. The last task assigned to the marked PRF is called the *event task* and it is used to update the ready queue. Each time an event task is selected, the PRF is unmarked. Hence, the event task cannot be used to update the ready queue a second time. A processor is marked each time a new task is assigned to it and unmarked each time an event task is chosen.

The `Update_R_Queue` routine updates the ready queue by inserting a new ready task chosen after the assignment of an event task. The new ready task is then selected by decrementing the number of immediate predecessors of the immediate successor tasks of the event task is by one. If the number is zero, that immediate successor task is chosen as the new ready task and is inserted into the ready queue. All immediate successors of the event task are then decremented. The new ready tasks are placed in the ready queue according to their priority -- maintaining the order of the queue.

A task is assigned by getting a ready task from the front of the ready queue. Thus, the task with the highest level and the maximum number of immediate successors is assigned first. Next, the processor selection routine, `Locate_P`, is used to select a processor, and the `Assign_Task` routine assigns the task to the selected processor.

The routine `Locate_P` selects the assigned processor that has the earliest start time for each assigned task considering the communication delay. The routine `Assign_Task` assigns a task to the selected processors at its start time calculated by `Locate_P`. Changes in these two routines are needed to improve the general heuristic, as we will show in subsequent sections. The heuristic is given in Algorithm 2.

Algorithm 2.

Input:

- Precedence graph, $H = (T, E_p)$
- m, number of processors in the parallel system

Output:

- GC, Gantt chart consisting of an array $\{1, \ldots, m\}$ for each processor in the system (list of tasks ordered by their execution time on a processor, including task start time and finish time)

```
begin
   level_graph(H)           Calculate level number for each task in H
   Init_Gantt(H)            Reset Gantt chart of each processor to nil
   Init_R_Queue(RQ)         Insert all tasks having no immediate predecessors
                            into the ready queue, in order by their level number
   Get_Task(AN, RQ)         Get AN, assigned task, from front of the Ready Queue
   Assign_Task(AN, 0, GC, 1, RQ)     Assigns AN to processor 1 at time 0
   repeat
      Update_R_Queue(RQ,AN,H)        Update ready queue using assigned task
      if RQ not empty
         Get_Task(AN,RQ)             Get next task from ready queue
         Locate_P(AN,GC,P_L,ST)      Schedule on processor with earliest ST
         Assign_Task(AN,ST,GC,PL,RQ)
   until all tasks in H are assigned
end
```

D. THE MAPPING HEURISTIC (MH)

The *mapping heuristic* (MH) proposed by El-Rewini and Lewis [14] is a modified list-scheduling technique. MH considers several real-world parameters that are neglected in the original list-scheduling heuristics covered in the previous section. MH models several target machine parameters such as interconnection topology, processor speed, link transfer rate, and contention delays. Like other list schedulers, MH uses the level of each node in the task graph as its level priority. But since communication is considered and processors may have different speeds, we face the problem of level alteration discussed earlier. In MH, the user is given the choice of whether communication cost is included in calculating the level of each node.

MH uses the general model described in Chapter II to model the parallel program and the target machine. In MH, we assume that the transmission rate over a link connecting any two adjacent processors is the same and equals R. The time to initiate message passing is the same for all processors and is equal to I. Recall also that T_{ij} is the execution time of task i when executed on processor j and $C(i_1, i_2, j_1, j_2)$ is the communication delay between tasks i_1 and i_2 when they are executed on processing elements j_1 and j_2 respectively. MH uses the given system parameters to compute the execution time and the communication delay as follows:

$$T_{ij} = \frac{A_i}{S_j} + B_j$$

$$C(i_1, i_2, j_1, j_2) = \left(\frac{D_{i_1 i_2}}{R} + I\right) \cdot H_{j_1 j_2} + CD_{j_1 j_2}$$

where $H_{j_1 j_2}$ represents the number of hops between processors j_1 and j_2, and $CD_{j_1 j_2}$ is the contention delay on the route from j_1 to j_2. In the following section, we discuss the routing tables in which we maintain the estimated values of $H_{j_1 j_2}$ and $CD_{j_1 j_2}$.

MH schedules the task with the highest level first and ties are broken in favor of the task with the largest number of immediate successors in the task graph. When a task is ready, i.e. all its predecessors have been scheduled, it is scheduled on the processor with the earliest finish time. The finish time of a task is determined by considering the following:

1. Processor speed
2. Link transfer rate

3. Message passing route
4. Number of hops
5. Delay due to contention

The details of how to compute the finish time of task *t* on processor *p* are given as follows:

There are four events that need to be considered by MH:

1. a task is ready
2. a task is done
3. a message is sent
4. a message is received

A task becomes ready as soon as its last unfinished immediate predecessor completes execution. When a task is ready, it can be scheduled for execution. When a task finishes execution on its assigned processor, the number of unfinished immediate predecessors of its immediate successors is decremented. When the number of unfinished immediate predecessors of a task becomes zero, the task is ready. When a message is sent, the route from the source to the destination becomes busy, carrying the message for a certain amount of time. Similarly, when a message is received, the route becomes free. When either of these events take place, the status of the machine interconnection network will have to be updated. This task is accomplished by updating the routing tables that we will examine next.

MH takes two inputs:

1. the parallel program task graph
2. the description of the target machine:
    - the number of processors
    - the interconnection network
    - the speed of each processor
    - the link transfer rate
    - the message passing initiation cost

It constructs and maintains routing tables to hold contention information. MH uses an event list to handle the four events given above. The high level description of the MH scheduling technique is given in Algorithm 3.

Algorithm 3.

Input:

- Precedence graph, $H = (T, E_p)$
- m, number of processors in the parallel system

Output:

- GC, Gantt chart consisting of an array $\{1, \ldots, m\}$ for each processor in the system (list of tasks ordered by their execution time on a processor, including task start time and finish time)

```
begin
   level_graph(H)          Calculate level number for each task in H
   Init_Gantt(H)           Reset Gantt chart of each processor to nil
   Init_R_Queue(RQ)        Insert all tasks having no immediate predecessors
                           into the ready queue, in order by their level number
   Get_Task(AN, RQ)        Get AN, assigned task, from front of the Ready Queue
   Assign_Task(AN, 0, GC, 1, RQ)     Assigns AN to processor 1 at time 0
   repeat
      Update_R_Queue(RQ,AN,H)        Update ready queue using assigned task
      if RQ not empty
         Get_Task(AN,RQ)             Get next task from ready queue
         Locate_P(AN,GC,$P_L$,ST)    Schedule on processor with earliest ST
         Assign_Task(AN,ST,GC,PL,RQ)
   until all tasks in H are assigned
end
```

1. Routing Tables

Since a communication link can be shared by more than one message, the contention delay must be considered in estimating the communication time. There are two time delays that contribute to communication delay:

1. the time delay incurred in transmitting data over an empty route
2. the queuing delay due to multiple messages sent through the same route (contention delay).

To compute the delay due to contention for each processor in the system, MH maintains a routing table that has contention information indexed by all other processors. In each processor's table, there is an entry for every other processor that contains three parts:

1. the number of hops H to reach the processor
2. the preferred outgoing line to use for that destination L
3. the communication delay due to contention CD

Initially, MH uses the shortest path between any two processing elements to determine the number of hops and the preferred outgoing tine. If there is more than one shortest path, a path is selected arbitrarily. Initially, the contention delay is zero. We use H_{ij} to refer to the number of hops between processors P_i and P_j that could be longer than the shortest path. L_{ij} and CD_{ij} are used similarly.

The routing tables are used to obtain the best route to send a message, and to compute the contention delay that guides the selection of the best processor for a certain task. These tables are updated during scheduling so the decisions made in choosing a route to send a message or in selecting a processor to run a task are based on the information describing the current traffic. Clearly, the more often the tables are updated, the more accurate the view of the current traffic will be. Thus, there is a trade-off between having a real view of the traffic and the complexity of the update procedure. MH updates the tables when a task starts sending a message to another task on a different processor, and when a message arrives at its destination. These two cases are important because the first causes a route to be busy for a certain amount of time, while the second frees a route for a subsequent transmission. Only the tables associated with processing elements where the traffic status has changed should be updated. The tables of the processors on the communication route are updated first using the `Direct_Update` routine. Then, the tables of the neighboring processors that have been affected and have not yet been updated are updated using the `Indirect_Update` routine.

The update routines used in MH operate as follows. Suppose that task t_1 on processor P_1 sends a message to task t_2 on processor P_2. In this case, the tables of the processors on the route from P_1 to P_2 should be updated. The table associated with processing element P_2 is not updated because the route from P_2 to other processors has not been affected. Similarly, if t_2 on processor P_2 receives a message from t_1 on processor P_1, all of the tables on the route from P_1 to P_2 except P_2 are updated.

IV. SCHEDULING NON-DETERMINISTIC TASK GRAPHS

A. BACKGROUND

Recall that several restricted versions of the problem of scheduling deterministic task graphs are computationally intractable. In order to obtain a static schedule for a given non-deterministic task graph the following procedure can be used:

1. Generate all possible execution instances of the parallel program that contains branching.
2. Construct a deterministic task graph for each execution instance.
3. Obtain a schedule for each instance using one of the deterministic scheduling heuristics presented in the literature.
4. Merge all these schedules into one unified schedule that preserves all the precedence relations of the precedence graph.

What is the major problem with this procedure? In the worst case there are too many execution instances for a non-deterministic parallel program. For this reason, a method needs to be found to reduce the amount of non-determinism in the parallel program, i.e. reduce the number of possible execution instances before applying the multi-phase approach.

B. REDUCING THE DEGREE OF NON-DETERMINISM

In practical applications, we noticed that the alternatives of several conditional branches might have many features in common. These common features include the way in which the alternative tasks communicate with other tasks in the program. Also. the amount of computation needed by the alternative tasks may be comparable.We exploit this property in order to reduce. or hopefully remove the non-determinism associated with conditional branching. Our ultimate goal is to reduce the probabilistic representation of the parallel program into a deterministic one, if possible. This can only be achieved in some special cases where all alternatives in the branch graph are very similar. However, in the general case, the basic idea is to find some tasks with common properties in order to be able to represent several tasks by only one single task. These properties express the way each task is related to other tasks in the program. A task can be related to other tasks in several ways, e.g., a precedence relation, data dependency. and occurrence in some execution instances. In order to achieve the goal of representing several tasks by only one task, they have to be very similar. In other words, these tasks should have the same relationships with the rest of the task in both the branch graph, and the precedence graph.

1. Similarity in Non-deterministic Task Graphs

Fact 1. Given a branch graph $G = (T, E_b)$, and $u, v \in T$, if u and v are similar, then there exists no execution instance that contains both of them.

Fact 2. Given a branch graph, $G = (T, E_b)$, and $u, v \in T$, if u and v are identical, then an optimal schedule for an execution instance EI_u, containing u can be obtained by replacing v by u in an optimal schedule for an execution instance, EI_v, containing v, where $(EI_u \cup EI_v) - (EI_u \cap EI_v) = \{u, v\}$

Recall that the main goal of the graph theoretic step is to reduce the degree of non-determinism in the task graph and to construct a reduced graph model from the original parallel program model. Although the following lemma deals with a rather restricted type of task graph, it inspires the concept of reducing the non-determinism in parallel programs.

LEMMA 1. Given a branch graph $G = (T, Eb)$, if every pair of similar nodes are identical, the branch graph can be reduced to exactly one execution instance, called the *representative execution instance* (REI). An optimal schedule for the tasks contained in any execution instance can he obtained from an optimal schedule of the tasks contained in the REI.

Proof. The *REI* can be constructed from the branch graph by removing all nodes, except one, from every set of identical nodes. The task graph that corresponds to the obtained REI is isomorphic to the task graph that corresponds to any possible execution instance of the program. Consequently, using the same scheduling technique, all possible execution instances are identical. Given the optimal schedule of the tasks of the REI, we can construct an optimal schedule for other tasks as follows. If task $v \in REI$ is scheduled on processor p, then any identical node of v will be assigned to the same processor p with the same execution order. Recall that two identical nodes cannot belong to the same execution instance (Fact I).

2. The Reduced Task Graph

It follows from Lemma 1 that if every set of similar nodes are identical, then non-determinism can be removed completely from the parallel program. If only some of the similar nodes are identical. then a reduced task graph may be constructed from the original task graph by replacing every set of identical nodes by a single representative node. In spite of the fact that this idea may be useful in reducing the amount of nondeterminism in parallel programs, it will not be very effective in general. The requirement of being identical is too restricted and will not be satisfied in most of the practical parallel programs It may be more effective if we relax this concept to allow some dissimilarities among the set of similar nodes.

There are three different types of possible dissimilarities among the alternative tasks of a branching statement:

1. the set of successors and predecessors in the precedence graph.
2. the cost of communication between the tasks and any common successor or predecessor in the precedence graph.
3. the amount of computation needed for execution.

Using these three measures. we can assume that a set of similar nodes are *almost* identical if the differences among its nodes are within some predetermined tolerance parameters.

The preprocessing step is then controlled by three parameters, α, β, γ which we call the *tolerance parameters*. When the amount of dissimilarities for a set of alternatives is bounded by the values of α, β, γ, these alternatives will be treated as if they are identical. The formal meaning of the tolerance parameters is described in Definition 6.

Definition 6. Given the tolerance parameters α, β, γ, two nodes u and v are said to be *almost identical* if it satisfies the following conditions:

- $max(|IMP(u) - IMP(v)|, |IMP(v) - IMP(u)|,$ $|IMS(u) - IMS(v)|, |IMS(v) - IMS(u)|) \leq \alpha$
- $abs(D(x, u) - D(x, v)) \leq \beta \ \forall x$ where (x,u) and $(x,v) \in E_p$

  $abs(D(u, y) - D(v, y)) \leq \beta \ \forall y$ where (u,y) and $(v,y) \in E_p$
- $abs(INS(u) - INS(v)) \leq \gamma$

In other words, the tolerance parameters provide bounds on the differences between two similar nodes to be treated as identical nodes. The parameter α provides a bound on the difference between the number of children and the number of parents of the two similar nodes. The parameter provides a bound on the difference between the data size transferred to and from the two nodes. Finally γ provides a bound on the difference between the required execution time of the two nodes.

It follows that the tolerance parameters, which are specified by the parallel program designer, determine the extent to which the identical relationship will be forced on similar nodes that are not identical. Each set of almost identical nodes will then be replaced by a single node in the program model -- the branch graph and the precedence graph. This process will result in another, hopefully smaller version of the program model denoted by the reduced program model. The reduced program model consists of the two directed graphs -- the *reduced branch graph* and the *reduced precedence graph.* The multi-phase technique presented in Section C will then be applied to the reduced graph model of the program, which usually contains a smaller number of alternatives, and in turn a smaller number of possible execution instances, and thus a lesser degree of non-determinism.

Algorithm 1 describes how to obtain the reduced program model from the original program model.

It is worth mentioning that the second step of the algorithm, in which the maximal set of almost identical tasks is obtained, is a complicated step. This is mainly due to the fact that unlike the relation *identical*, the *almost identical* relation is not transitive. It can be shown that, given the value of the tolerance parameters, the problem of identifying the minimum number of almost identical sets of tasks is NP-hard. A simple greedy algorithm can be used to find some maximal sets of almost identical nodes such that the number of such sets is close to the optimal minimum number.

Algorithm 1.

Input:

- Branch graph, $G = (T, E_b)$
- Precedence graph, $H = (T, Ep)$
- Tolerance parameters α, β, γ

Output:

- Reduced branch graph, $G1 = (G1, E_{rb})$
- Reduced precedence graph, $H1 = (T1, E_{rp})$

```
begin
    read the values of the tolerance parameters $\alpha, \beta, \gamma$.
    determine all maximal sets of almost identical nodes
    for every set S of almost identical nodes in T and $|S| \geq 2$ do
    begin
        let a be the parent of any node in S
        let t be the child of any node in S
        modify the branch graph as follows
        begin
            $T1 = T \cup \{x\}$ add a single node x that represents all nodes in S
            $E_{rb} = E_b \cup \{(s,x),(x,t)\}$
            $P(u,x) = \sum P(s,y) \; \forall y \in s$
            $P(x,t) = 1$
            $T1 = T - S$ remove all nodes of S along with their incident edges
        end
        modify the precedence graph as follows
        begin
            $T1 = T \cup \{x\}$   add a single node x that represents all nodes in S
            $INS(x) = max(INS(y))\ \forall y \in s$
            $E_{rp} \rightarrow E_p \cup \{(u,x)\}$ where $u$ is a parent of any node $y \in S$
            $D(u,x) = max(D(u,y))\ \forall y \in s$   for any node $y \in S$
            $E_{rp} = E_p \cup \{(x,v)\}$ where v is a child of any node $y \in S$
            $D(u,v) = max(D(y,v))\ \forall y \in s$   for any node $y \in S$
            $T1 = T - S$   remove all nodes of S along with their incident edges
        end
    end for
end
```

Example 1. We apply Algorithm I on the program model given in Figures 3 and 4. We assume that the tolerance parameters, α, β, γ, are given the values 1, 5, and 10, respectively. Studying the program model. one can see that there are only two non-singleton similar sets of tasks, *{2, 3}* and *{6, 7, 8}*. Comparing the dissimilarities of all possible subsets of similar nodes with the tolerance parameters. We can reduce nodes *2* and *3* to a single node *23* with execution time equaling 12. Using the above tolerance values. only nodes 2 and 3 may be reduced to one node. The value of *D(5, 23)* will be assigned the value of *max(D(5, 2), D(S, 3))* which is equal to 15 in order to guarantee the feasibility of the final schedule. The reduced program model is shown in Figure 3. Not that there are only three possible execution instances of the reduced model. compared to six instances in the original model.

C. THE MULTI-PHASE APPROACH

In this step, a heuristic algorithm is applied to the reduced task graph in order to obtain a static schedule for the original non-deterministic task graph. Several branch-free (deterministic) task graphs are generated to represent all execution instances of the parallel program. Each task graph is scheduled using one of the static scheduling techniques that can deal with branch-free graphs, and all the generated schedules are merged into a unified one In our experiments we used the MH algorithm to obtain a static schedule for each execution instance. If the number of possible execution instances is not too large, all possible instances are generated and the corresponding task graphs are scheduled. However, if the number of possible instances is exponential, then some of the execution instances are considered. In this case, we select the execution instances that are among the most likely ones and cover all the program tasks. i.e. each task will be included in at least one execution instance. The details of the generation process are given in Section IV.1.

This algorithm is divided into four phases:

1. Generate some (all) possible execution instances of the branch graph.
2. For each execution instance, the corresponding task graph is constructed.
3. A schedule for each constructed task graph is obtained in the form of a Gantt chart.
4. A unified schedule is found by merging all the charts obtained in phase 3 The following procedure summarizes the whole process The details of the generation process are given in Sections 1 - 4.

```
procedure MPA
begin
    Generate     Generate some possible execution instances
    Construct    Construct a task graph for each generated execution instance
    Schedule     Produce a schedule for each constructed task graph
    Merge        Merge all produced schedules into a unified one
end
```

1. Phase 1: Generate

To schedule task graphs with branches, one might need to consider all possible execution instances of the reduced program model. In general, the number of possible execution instances of a program can be exponential -- and considering all the possibilities could be computationally expensive. However, considering all the possible execution instances might be feasible in some cases. particularly when the number of all instances is polynomial. A typical example for such a case is *bounded-height branch* graphs. Bounded-height branch graphs are the graphs in which the height of each component is bounded by a constant h. The number of possible execution instances in such a graph is polynomial and of order $O(cb^h)$, where c is the number of connected components of the branch graph and b is the maximum number of children of any node in the graph. In these special cases. all possible instances are generated and the corresponding task graphs are scheduled. In general, when the number of execution instances is exponential, we introduce an algorithm that generates a smaller number of execution instances which cover all the tasks in the program. The selected instances are the ones which have the highest probability, with the restriction that each program task is included in at least one instance.

Given a branch graph $G = (T, E_b)$, we introduce an algorithm that finds the most likely set of execution instances $SetEI = \{EI_1, EI_2, ..., EI_m\}$ that covers all program tasks. Each execution instance EI_i. consists of a set of tasks $\{v_1, v_2, ..., v_{ki}\}$, where k_i is the number of tasks included in instance EI. It can be observed that $SetEI$ covers the set T iff

$$\bigcup_{i=1}^{m} EI_i = T.$$

The algorithm is partitioned into two parts: evaluating the task probability $\mu(t)$ for every task t and generating the set of execution instances $SetEI$. In this algorithm a set of execution instances that cover all the tasks is found. Every task t, with $\mu(t) = 1$ is included in all the execution instances. The most likely execution instance, which is the instance with the highest probability, is considered first, and all the nodes in that instance are marked. We then consider the most likely execution in- stance among the unmarked nodes and this process is repeated until all nodes are marked. The details are given in Algorithm 2.

Algorithm 2.

Input:

- Branch graph, $G = (T, E_b)$
- Precedence graph, $H = (T, E_p)$

Output:

- Set of execution instances, *SetEI*

```
begin
    Compute the probability that a task $t$ will be executed $\mu(t)$
    forall $t \in T$ do
    begin
        let $IMP(t) = \{v1, v2, ..., vm\}$ be the set of immediate predecesors of
            $t$ in $G$
        if $IMP = \Phi$ then $\mu(t) \leftarrow 1$        $t$ is a source node
        else $\mu(t) \leftarrow \sum_{i=1}^{m} P(v_i,t) \cdot \mu(v_i)$
    end
    Generate the set of execution instances with highest probability that
        covers all tasks
    $SetEI \leftarrow \Phi$
    repeat
        $EI \leftarrow \Phi$
        for every connected compnent in the Branch Graph do
        begin
            let $t$ be the source node
            $EI \leftarrow EI \cup \{t\}$
            repeat
                let $IMS(t)$ be the set of all immediate successors of $t$
                if all elements in $IMS(t)$ are marked then
                    $EI \leftarrow EI \cup \{t\}$ , where $\mu(u) \geq \mu(x)\ \forall x \in IMS(t)$
                    $EI \leftarrow EI \cup \{t\}$ ,where $\mu(u) \geq \mu(x)$
                                                $\forall (unmarked) x \in IMS(t)$
                mark $u$
                $t \leftarrow u$
            until $IMS(t) = \Phi$
        end
        $SetEI \leftarrow SetEI \cup EI$
    until all nodes are marked
end
```

2. Phase 2: Construct

In this phase, we construct a precedence task graph for each execution instance generated in Phase 1. Each task graph consists of the nodes given in the corresponding execution instance and the precedence relations among them. It also shows the amount of computation needed at each node as well as the size of the data messages passed among the nodes. For each generated execution instance *EI*, an instance task graph $ITG = (EI, E_{p_s})$ gives the tasks and their precedence relations in the execution instance where $EI \subseteq T$ and $E_{p_s} \subseteq E_p$. Recall that *Ep* is the set of edges in the precedence graph $H = (T, E_p)$. An edge $(u,v) \in E_p$ if $u, v \in EI$ and $(u,v) \in E_p$. The values of the parameters *INS(*)* and *D(*, *)* remain the same as given in the original precedence graph. The steps to produce a set of instance task graphs, denoted by *SetITG*, are given in Algorithm 3.

Algorithm 3.

Input:

- set of execution instances, *SetEI*
- precedence graph, $H = (T, E_p)$

Output:

- set of instance task graphs, *SetITG*

```
begin
    Generate a set of instance task graphs
    $SetEI \leftarrow \Phi$
    for every generated execution instance $EI$ do
    begin
        $E_{p_s} \leftarrow \Phi$
        $E_{p_s} \leftarrow E_{p_s} \cup (u,v)$ for every pair $(u, v) \in EI$ and $(u, v) \in E_p$
        let $ITG = (EI, E_{p_s})$
        $SetITG \leftarrow SetITG \cup ITG$
    end
```

3. Phase 3: Schedule

Each task graph generated in Phase 2 can be scheduled independently using a static deterministic scheduler. In our system, we use the MH static scheduling heuristic that was introduced by El-Rewini. MH takes a task graph and a target machine description as its input and produces a schedule in the form of a Gantt chart. Communication cost and target machine topology are considered in making scheduling decisions. Given an instance task graph and a target machine description, a schedule in the form of a Gantt chart will be fed to the merge algorithm, described in Phase 4, in order to obtain a unified schedule. The information given in any Gantt chart can be represented using two functions $P(t)$ and $O(t)$. For each task t, the functions $P(t)$ and $O(t)$ indicate the processor assigned to task t and its order of execution, respectively.

The output of this phase is a set of k Gantt charts resulting from scheduling k different instance task graphs of the program. Each Gantt chart g is expressed in terms of the functions $Pg(t)$ and $Og(t)$ for each task t. Having $Pg(t) = -1$ implies that t does not belong to the corresponding execution instance EI_g. Associated with each Gantt chart g is the probability of the occurrence of EIg, denoted by $prob(g)$. The probability of occurrence of EIg can be computed as follows: $prob(g) \leftarrow \prod P(u,v)$ for all $(u,v) \in E_b$ and $u, v \in EI_g$. Algorithm 4 shows how to produce a set of Gantt charts, denoted by $SetIGC$.

Algorithm 4.

Input:

- Set of instance task graphs, *SetITG*
- Branch graph, $G = (T, E_b)$

Output:

- Set of Gantt charts, *SetGC*. (Formally, $P_g(t), O_g(t), 1 \le g \le |SetGC|$)
- The probability of occurrence, $prob(g), 1 \le g \le |SetGC|$

```
begin
    $SetGC \leftarrow \Phi$
    for g = 1 to $|SetITG|$ do
    begin
        Generate the probability of occurrence
        $prob(g) \leftarrow \prod P(u,v), \forall (u,v) \in E_b$ and $u, v \in taskgraph\ G$
        Generate a schedule for every task graph
        $(P_g(t), O_g(t)) \leftarrow Schedule(taskgraph\ G)$    using branch-free
                                                              scheduling technique
        $SetGC \leftarrow SetGC \cup (P_g(t), O_g(t))$
    end

end
```

4. Phase 4: Merge

In this phase, a number of Gantt charts are combined into a unified schedule. The schedule is given in the form of processor allocation and execution order of the tasks allocated to the same processor. The merge algorithm produces a unified schedule represented by $P(t)$ and $O(t)$ for each task t. The functions $P(t)$ and $O(t)$ are computed using $Pg(t)$, $Og(t)$, and $prob(g)$, $1 \le g \le |SetGC|$.

The merge algorithm consists of two steps: task allocation to compute $P(t)$ and task ordering to determine $O(t)$, $1 \le t \le M$. The allocation of each task is obtained by considering all Gantt charts as shown in Algorithm 5. We use the summation of the probabilities, as well as the number of times a task is assigned to the same processor, to determine the allocation of that task. The execution order of each pair of nodes assigned to the same processor is obtained by considering all the Gantt charts in which both tasks are assigned to the same processor. In the event when there are two different orders for the two tasks, a weighted majority function is used to determine the order in the unified schedule. The weighted majority function is defined as the summation of the probability of occurrence as shown in Algorithm 5. If the two nodes do not appear on the same processor in any Gantt chart, their order is determined by considering their precedence relation, if any, in the original precedence graph. Otherwise, the order is determined randomly. The detailed steps that determine the allocation and the order of execution in the unified schedule are given in Algorithm 5.

Algorithm 5.

Input:

- Set of Gantt charts, *SetGC*. (Formally, $P_g(t)$, $O_g(t)$, $1 \le g \le |SetGC|$)

- The probability of occurrence, *prob(g)*, $1 \le g \le |SetGC|$

Output:

- A unified schedule *P(t)* and *O(t)*

```
Task Allocation
Let $A_p(t)$ = number of times that $P_g(t) = p$,  $1 \le g \le |SetGC|$
Let Bp(t) = $\sum_{g=1}^{|SetGC|} prob(g)$ , when $P_g(t) = p$

  $P(t) = \{j | ((B_j(t), A_j(t)) \ge (B_I(t), A_I(t)), (1 \le I \le N))\}$

Task Ordering
for $p = 1$ to $N$ do
    for each pair $(t_i, t_j)$ such that $P(t_i) = P(t_j) = p$ do
    begin
        $weight \leftarrow 0$
        for $g = 1$ to $|SetGC|$ do
            if $P_g(t_i) = P_g(t_j) = p$ then
                if $O_g(t_i) < O_g(t_j)$ then
                    $weight \leftarrow weight - prob(g)$
                else
                    $weight \leftarrow weight - prob(g)$
                if $(weight > 0)$ or $((t_i, t_j) \in E_p)$ then
                    $t_i <_p t_j$            $<_p$ is the order on p
                elseif (weight < 0) or $((t_i, t_j) \in E_p)$ then
                    $t_j <_p t_i$
                else
                                             Order is picked randomly
            end
        end
end
```

Example 2. We consider the reduced program model given in Figure 1. Since non-singleton similar sets still exist after applying the graph theoretic approach, all four phases of the multi-phase approach should now be applied on the reduced program model. The output of each phase of the four phases is given below.

Output of Phase 1. It can be observed that the reduced program model has only three execution instances as opposed to six in the original model. If we assume that the height of the branch graph is bounded, then all possible execution instances are generated as follows. *EIi* is the set of tasks forming execution instance *i*.

$$EI_1 = \{1, 23, 4, 5, 6, 9\}$$

$$EI_2 = \{1, 23, 4, 5, 7, 9\}$$

$$EI_3 = \{1, 23, 4, 5, 8, 9\}$$

The probability of occurrence for the execution instances are

$$prob(EI_1) = 0.7$$

$$prob(EI_2) = 0.2$$

$$prob(EI_3) = 0.1$$

Output of Phase 2. In this phase. we construct an instance task graph for each execution instance generated in Phase 1. The task graphs are given in Figures 7-9

Output of Phase 3. A Gantt chart for each instance task graph is obtained using a branch-free scheduling technique. Figures 6-8 show all instance task graphs and their corresponding Gantt charts.

Output of Phase 4. According to Algorithm 5, given the probability of occurrence of each execution instance and the corresponding Gantt chart, a unified schedule is obtained as follows.

• Allocation:

$P(1) = P_1$	$P(5) = P_2$	$P(8) = P_1$
$P(23) = P_2$	$P(6) = P_1$	$P(9) = P_1$
$P(4) = P_2$	$P(7) = P_1$	

• Order:

Processor P_1	**Processor P_2**
O(1) = 1	O(5) = 1
O(6) = 2	O(23) = 2
O(7) = 3	O(4) = 3
O(8) = 4	
O(9) = 5	

Note that the order between tasks that do not occur in the same execution instance is meaningless.

V. THE PROBABILISTIC MERGE HEURISTIC

A. DESIGN CONSIDERATIONS

In chapter IV, we examined two techniques proposed by El-Rewini and Ali to statically schedule non-deterministic task graphs. In the first approach, an attempt is made to reduce the degree of non-determinism by exploiting similarities among nodes in the same execution instance. Yet we have found that, unless the tolerance parameters α, β, γ are quite large, little reduction of the task graph occurs. We should also consider that unless *all* non-determinism is removed from the precedence graph, we are still faced with the problems of statically scheduling non-deterministic task graphs. While the *reduction phase* approach has definite merits as a pre-processing step for further scheduling techniques, we find that it provides insufficient benefits as a solitary scheduling algorithm.

The second approach to scheduling non-deterministic task graphs was the *multi-phase* scheduling algorithm. In this approach, all possible execution instances are generated, individually scheduled with MH, and then merged into a unified schedule using a weighted majority function. This approach produces acceptable scheduling results on a wide variety of test cases. As is often the case with scheduling heuristics, however, there is a great computational expense in applying the multi-phase approach. First, consider that the multi-phase algorithm (as well as the reduction-phase algorithm) relies heavily on the generation of all probabilistic execution instances from the branch graph. In general, however, the number of execution instances in a non-deterministic branch graph is exponential with respect to the number of constituent tasks. El-Rewini and Ali consider a special case, the *bounded-height branch graph*, in which the height of the task graph is bounded by a constant *h*. The number of possible execution instances in such a graph is polynomial of order $O(cb^h)$, where *c* is the number of connected components of the branch graph (number of tasks), and *b* is the maximum number of children at each node. It should be obvious that a scheduling algorithm containing a step with a complexity of $O(cb^h)$ will not be scalable to a parallel programs comprising a large number of tasks. As shown in

Figure 2, the time to compute the multi-phase approach quickly becomes unmanageable, as the number of nodes in the branch graph is increased. El-Rewini and Ali suggest that one should select a set of execution instances that are among the most likely ones and cover all the tasks, i.e., each task will be included in at least one execution instance. Yet this step, which involves searching an arbitrary tree structure for the *n* highest probability paths, is known to be NP-Complete [6]. It may be interesting to the reader that the actual *generation* of the execution instances comprises only a small portion of the computational complexity of the multi-phase approach. It is the *scheduling* of each execution instance that accounts for the majority of the computational requirements.

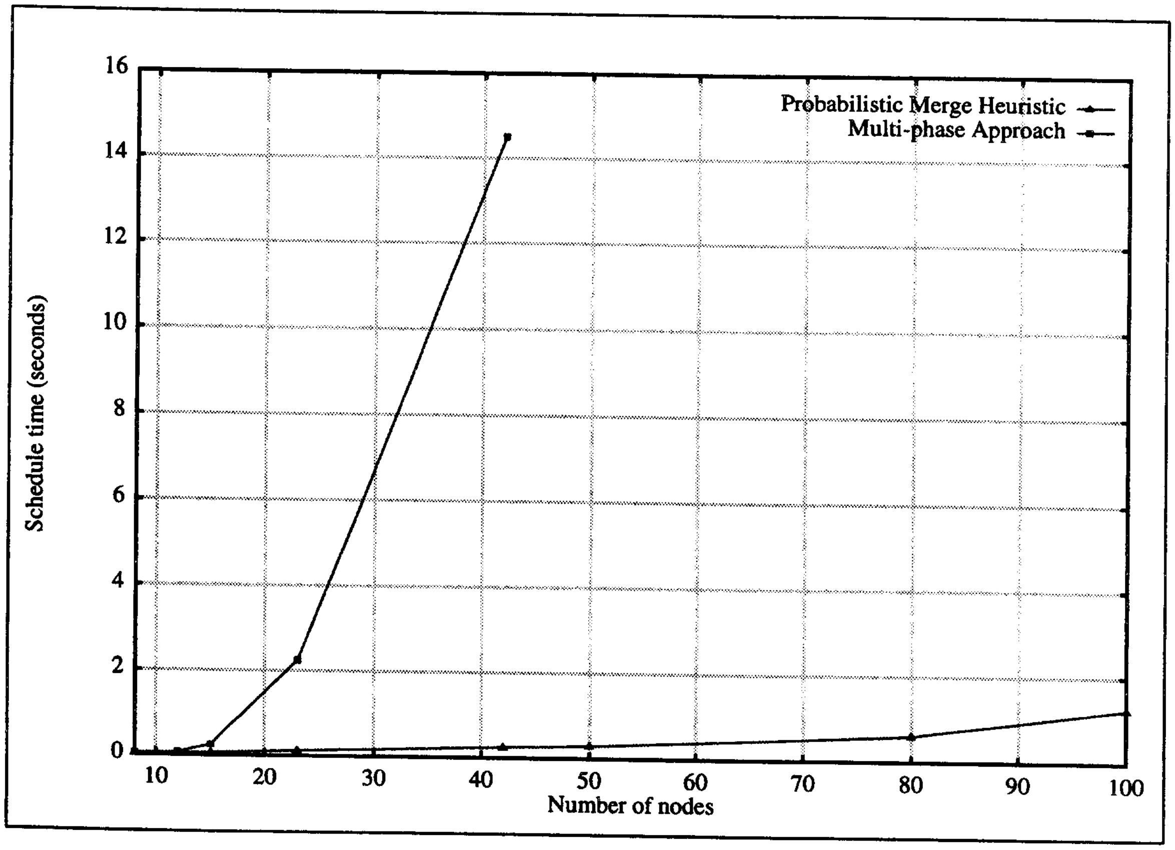

Figure 2: Scheduling Runtimes

B. THE PROBABILISTIC MERGE HEURISTIC

We propose an alternative approach to statically scheduling non-deterministic task graphs. While this approach can also benefit from the reduction phase, the algorithm does not require the generation of the execution instances from the branch graph. Our approach involves two fundamental modifications to the classical list scheduling algorithm. We have applied these modifications to the MH scheduling algorithm.

1. Level Calculation

The first phase of the probabilistic merge heuristic involves a modification of the task priority assignments of the classical list scheduling algorithm. The level of each node is computed as before, a function of the length of the critical path to that node and the sum of the communications along that path. However, we then weight the resulting priority by the conditional probability of the execution of the node. This will weight the ready queue, so that the highest probability tasks tend to be scheduled first, and lower probability tasks tend to be scheduled last. The principle behind this design decision is that often there is a single execution instance which has a very high probability, followed by other lower probability execution instances. We therefore optimize the schedule around this most probable execution instance, and allocate the less likely tasks to the remaining system resources.

2. Probabilistic Merging

As we generate the level assignments for each task, we record a list of all other nodes which are not in the same execution instance and have computational requirements *at least as large* as the current task. While it may seem that this step again requires the generation of all execution instances, it is implemented by generating the *transitive closure* of the branch graph, a function with a complexity of $O(|V| \cdot |E|)$, where V is the number of nodes, and E is the number of edges. To determine if two nodes exist in the same execution instance, one merely checks if an arc exists between the two nodes in the transitive closure.

We then schedule the tasks according to their level priority with the MH scheduling algorithm. Every time a task becomes ready, however, we first examine the list of nodes which are not in the same execution instance. If there is a node which is not in the same execution instance and has already been scheduled, we examine its suitability for *probabilistic merging*.

There are three criteria which must be met in order to merge two nodes in the Gantt chart, without violating the data dependencies in the precedence graph:

1. the nodes must not be members of the same execution instance
2. the candidate node must fit within the time slot of the node which has already been scheduled.
3. the start time of the previously scheduled node must not be earlier than the candidate node.

The first criteria ensures that the tasks must not be members of the same execution instance. This was guaranteed when the candidate list was originally generated, by using the transitive closure test. The second criteria requires the candidate node fit within the time slot of the node which has already been scheduled. Since level priority calculations include task size, the earlier scheduled tasks will tend to be larger than later tasks. Thus, the containment test will often succeed. The final criteria ensures that the start time of the first node must not be earlier than the second node. This is necessary to preserve data dependence relationships in the precedence graph.

The resulting algorithm traces down the list of nodes which are candidates for probabilistic merging, attempting to satisfy all three conditions. If no node matches these conditions, the current node is scheduled normally, occupying its own slot in the Gantt chart. If there is a previously scheduled node which matches these conditions, the current node is added to the list of possible execution instances for that time slot in the Gantt chart.

The subsequent schedule would be implemented on the target parallel computer by loading all conditionally executed tasks, which have been merged into a single slot on the Gantt chart, onto a single computational node. At runtime, a decision will be made as to which execution instance is active, and the corresponding task in the conditional task list will be executed.

The steps detailing the Probabilistic Merge Heuristic are shown in Algorithm 1.

Algorithm 1.

Input:

- Branch graph, $G = (T, E_b)$
- Precedence graph, $H = (T, E_p)$
- m, number of processors in the parallel system

Output:

- GC, Gantt chart consisting of an array *{1,..., m}* for each processor in the system (list of tasks ordered by their execution time on a processor, including task start time and finish time)

```
begin
   Plevel_graph(H)          Calculate level number for each task in H,
                               weighted by probability of execution
   Init_Gantt(H)            Reset Gantt chart of each processor to nil
   Init_R_Queue(RQ)         Insert all tasks having no immediate predecessors
                            into the ready queue, in order by their level number
   Get_Task(AN, RQ)         Get AN, assigned task, from front of the Ready Queue
   Assign_Task(AN, 0, GC, 1, RQ)      Assigns AN to processor 1 at time 0
   repeat
      Update_R_Queue(RQ,AN,H)       Update ready queue using assigned task
      if RQ not empty
         Get_Task(AN,RQ)            Get next task from ready queue
         Find_merge(AN,GC,P_L,ST)   Attempt to merge task with existing task
         Locate_P(AN,GC,P_L,ST)     Otherwise, schedule on processor with
                                       earliest ST
         Assign_Task(AN,ST,GC,PL,RQ)
   until all tasks in H are assigned
end
```

Figure 3 shows the schedules resulting from scheduling the example branch graph and precedence graph in Figure 1. Figure 3a shows the schedule generated by scheduling the branch graph and the precedence graph with the multi-phase approach. The resulting probabilistic schedule runs with a critical path of 44 seconds. Figure 3b shows the schedule from the same branch graph and precedence graph, with the level weighting phase of the probabilistic merge heuristic. The resulting probabilistic schedule runs with a critical path length of 40 seconds, an improvement of 10%. Figure 3c shows the final schedule after probabilistic task merging has been performed. This final schedule runs with a critical path length of 33 seconds -- an overall improvement of 33%. We would like to emphasize that we did not choose the example branch and precedence graphs to favor the performance of the probabilistic merge heuristic. In fact, the branch and precedence graphs were taken from El-Riwini's description of the multi-phase approach [1].

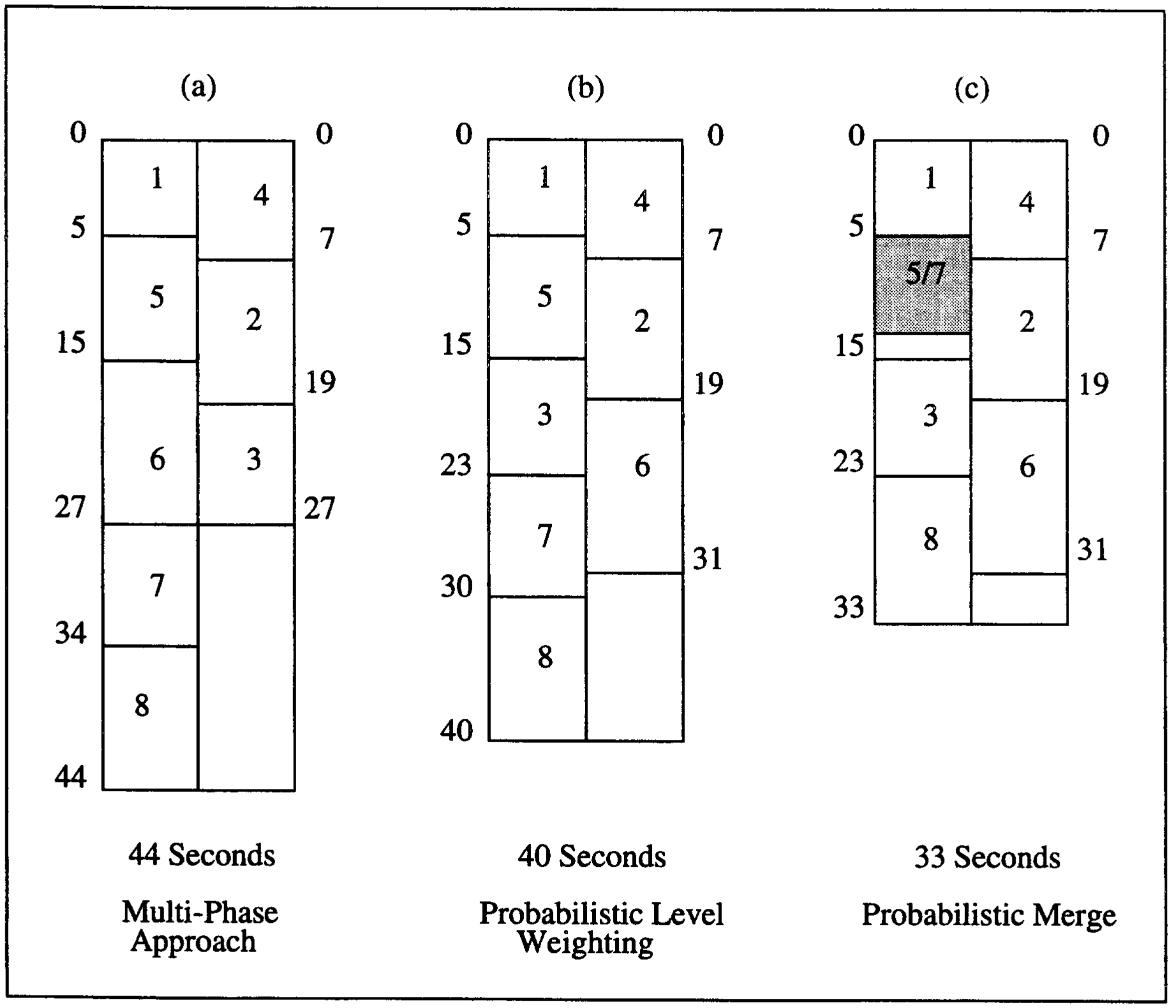

Figure 3: Resulting Schedules

C. SIMULATION RESULTS

In this section, we report a summary of the results of the experiments conducted to evaluate the performance of the proposed techniques using randomly generated parallel programs.

The comparison test consists of applying both heuristics to a wide range of randomly generated task graphs, with 10, 20, 40, 50, 80, and 100 tasks. We then compare the resulting schedule lengths from the multi-phase heuristic against the schedule length of the probabilistic merge heuristic. The number of processors is varied from 2 to 16. It is important to note that the schedule length of a non-deterministic task graph will be probabilistic in nature. This is due to the uncertainty when the scheduler is executed as to which path of each conditional branch in the task graph will be followed. Thus the schedule length is determined as the summation of the runtimes of the individual execution instances, each weighted by their probability of execution.

The input programs were classified by the parameters *CE* and *ND*. *CE* is the ratio of the combined communication of all tasks in the task graph over the combined computation of all tasks. In other words, the value of the parameter *CE* reflects the degree of communication versus computation in the input programs, where higher values of *CE* implies higher degrees of communication. *ND* is the ratio of the number of tasks with task probabilities less than one to the total number of tasks. In other words, the value of the parameter *ND* reflects the degree of non-determinism in the input programs, where higher values of *ND* implies higher degrees of non-determinism. For given values of *CE* and *ND*, several parallel programs were generated randomly with varied precedence graphs and task graphs.

1. Scheduling Efficiency

When comparing scheduling heuristics, one must consider not only the quality of the schedule, but the complexity of the algorithm as well. It can be seen in Figure 2 that the multi-phase heuristic does not scale well to a large number of tasks in the parallel program.

Despite the exponentially increasing runtimes of the multi-phase heuristic, we were surprised that execution time was not the limiting factor in this algorithm. In fact, data could not be collected for the multi-phase heuristic with large precedence graphs due to memory limitations. With more than 40 nodes, the multi-phase heuristic would exhaust virtual memory on our workstation. This is due to the enormous cost of generating all execution instances, and subsequently scheduling them with MH. The probabilistic merge heuristic, in contrast, scales according to a low-order polynomial function, and generates schedules with reasonable runtimes even with large numbers of tasks.

2. Schedule Runtimes

Finally, we compare the runtimes of schedules generated by the multi-phase heuristic against schedules generated by the probabilistic merge heuristic. For the set of sample branch graphs and precedence graphs we used the proposed techniques to generate the conditional schedule, and then calculated the probabilistic critical path length as the weighted sum of the path length of each execution instance. It can be seen in Figure 4 that the probabilistic merge heuristic consistently generated schedules which had shorter probabilistic critical path lengths. These critical paths of the probabilistic merge heuristic were, on average, 28% faster than those of the multi-phase heuristic.

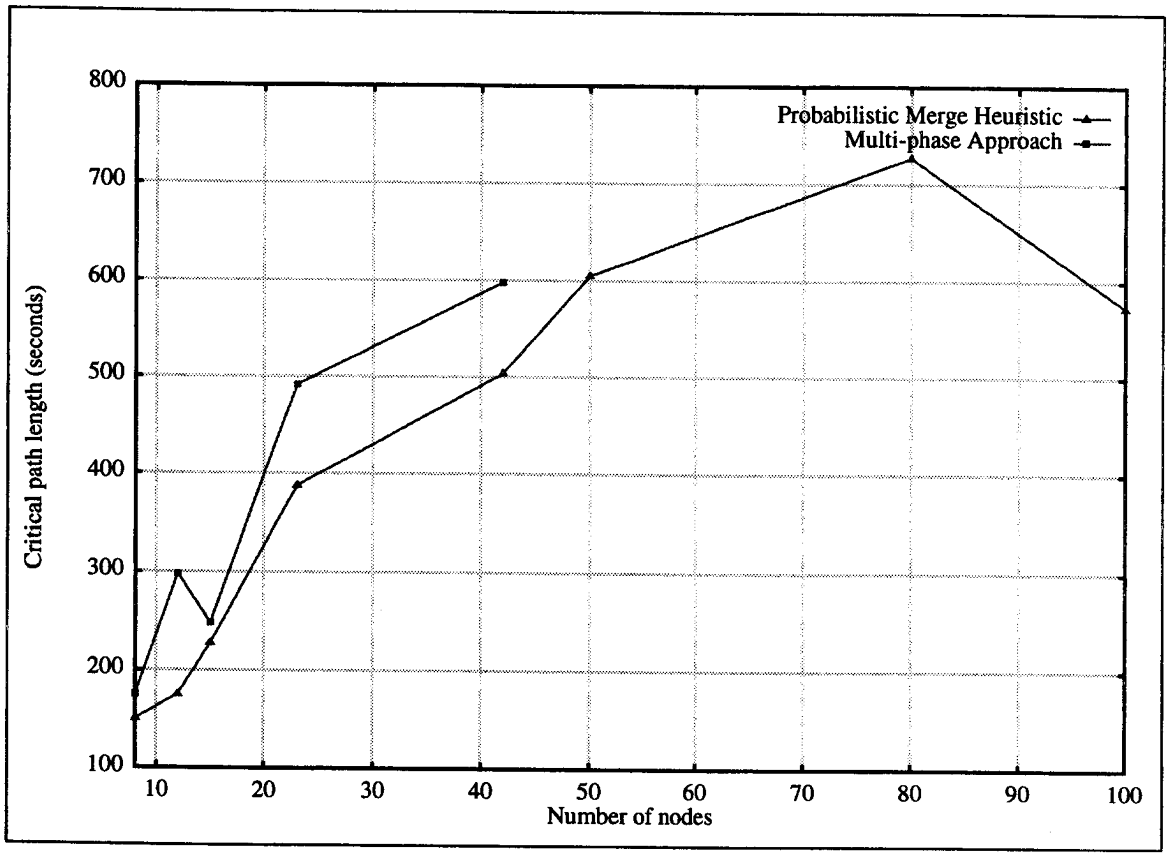

Figure 4: Critical Path Lengths

D. CONCLUSIONS

In this thesis, we have studied several approaches to the problem of statically scheduling tasks which comprise a non-deterministic task graph. We have reported a summary of the results of experiments conducted to evaluate the performance of the proposed techniques using randomly generated parallel programs.

The results of applying both El-Rewini's multi-phase heuristic and the probabilistic merge heuristic to a wide range of randomly generated task graphs, show that the probabilistic merge heuristic consistently generates schedules which have shorter probabilistic critical path lengths, despite significantly shorter runtimes. The critical paths of the probabilistic merge heuristic were, on average, 28% faster than those of the multi-phase heuristic.

Our future work includes the extension of the probabilistic merge heuristic to graphs which contain periodic tasks, such as loops. We feel there is sufficient data-flow information to optimize the merge phase, in a fashion similar to cylinder packing.

LIST OF REFERENCES

1. H. El-Rewini and H. Ali, Static Scheduling of Conditional Branches in Parallel Programs, *J. Parallel Distrib. Comput.* **24** (Jan 1995), 41-53.

2. T. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, Prentice Hall, Englewood Cliffs, 1992.

3. T. Casavant and J. Kuhl, *A Taxonomy of Scheduling in General Purpose Distributed Computing Systems*, IEEE Transactions on Software Engineering, Vol. SE-14, No. 2, February 1988.

4. H. El-Rewini, T. Lewis, H. Ali, *Task Scheduling in Parallel and Distributed Computing*, Prentice Hall, Englewood Cliffs, 1994.

5. E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.

6. M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

7. M Gonzalez, *Deterministic processor scheduling*, Comput. Surveys, 9, No. 3 (Sept. 1977).

8. T. Hu, Parallel sequencing and assembly line problems, *Operations Res.* **9** (1961), 841-848.

9. V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessing*, MIT Press, 1989.

10. D. Towsley, Allocating programs containing branches and loops within a multiple processor system, *IEEE Trans. Software Eng.* SE-12, No. 10 (Oct. 1986).

11. J. Ullman, NP-complete scheduling problems, *J. Comput. System Sci.* **10** (1975), 384-393.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center..2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library...2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code CS..1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. Theodore Lewis, Code CS/TL..1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Dr. David Pratt, Code CS/DP...1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. Mr. Robert George...2
   NSF ERC for Computational Field Simulation
   Mississippi State University
   Box 9627
   Mississippi State, MS 39762